A Model for Provably Secure Software Design

Alexander van den Berghe* Koen Yskout*, Wouter Joosen*
*imec-DistriNet
KU Leuven, 3001 Leuven, Belgium
firstname.lastname@cs.kuleuven.be

Riccardo Scandariato†
†Software Engineering Division
Chalmers and Göteborg University, 41756 Göteborg, Sweden
firstname.lastname@cse.gu.se

***Abstract*—Both academia and industry advocate the security by design principle to stress the importance of dealing with security from the earliest stages in software development. Nevertheless, designers often have to resort to their own knowledge and experience to pro-actively identify and mitigate potential security problems. Moreover, research shows that *correctly* applying security solutions is a much more significant challenge for designers, rather than finding an adequate solution. Therefore, there is a need for techniques that ensure a correct application of a security design solution. The contribution of this paper is a model in which the security-relevant aspects of a design can be precisely expressed in an integrated manner, enabling thorough reasoning about these aspects. We illustrate this model with a sizeable model of a banking system and show how the precise semantics of this model enables the tool-supported construction of proofs about the correctness of the applied design solutions. Our proposal thus enables designers to obtain stronger guarantees, ensuring the correctness of their solutions. The presented model can serve as the foundation for security by design, in time enabling automated security verification throughout the software development cycle.**

***Keywords*-Security by design, software design, security analysis**

I. Introduction

Security by design is often evoked to stress the importance of dealing with security early on in the software development life cycle, i.e. starting from the high-level design of a software system. In practice, this means that designers need to identify potential security problems early on, and pro-actively provide countermeasures preventing these. They typically have to rely to a large extent on their own knowledge and experience, augmented with a collection of common security design strategies such as security tactics [1] and patterns [2], [3], [4].

Earlier research has indicated that selecting an adequate countermeasure from such a collection is not the critical challenge for designers, but correctly applying the selected countermeasure is [5]. Thus designers could profit from support to *correctly instantiate* a known security design solution to improve the correctness and quality of their design artefacts.

The main focus of this paper is the proposal of a security model that allows to precisely model a wide variety of security solutions, and enables a formal verification of the instantiated security solutions. This aspect of the secure development process has not yet been thoroughly investigated [6].

We propose a model that is inspired by Data Flow Diagrams (DFD) [7]. DFDs were originally introduced as a general-purpose analysis tool, and are straightforward to use and understand. Furthermore, they are already extensively used in security design, for example to perform a security analysis with STRIDE [8, chapter 3]. In contrast to generic DFDs, though, our model has been specifically designed for the purpose of security. In practice, this means that our model replaces the traditional DFD element types with multiple security-specific ones, and attaches a precise semantics to each such element. These semantics can then be leveraged to specify security designs, and formally prove whether or not security properties, such as data confidentiality, hold for a design. Thereby complementing the designer's knowledge with strong, proven guarantees. The model and accompanying verification techniques are intended to serve as a foundation for security by design. We evaluate our proposal by modelling the security design of a realistically sized banking application.

II. Preliminaries

In this section, we lay the groundwork for this paper by (1) introducing the running example, (2) providing some background information about DFDs, and (3) illustrating why DFDs by themselves are not sufficient for security by design.

Running example. Throughout this paper, we demonstrate our approach with the design of a realistically complex and large banking system. We have already used this system in earlier work [5] and it has served as the project assignment for a master course on software architecture. The complete architecture is documented in a 32 page PDF file containing UML diagrams and documentation of its interfaces [9].

We have modelled the entire banking system in our model. Nevertheless, for comprehensibility, this paper focuses on only a part of it, namely customers logging in via the bank's website and initiating a transaction (wire transfer) between two accounts. An evident security requirement in this context is that an attacker cannot tamper with such transactions.

DFDs. As mentioned, our model is inspired by Data Flow Diagrams (DFD), which consist of (external) entities, processes, data stores, and data flows between these elements. Figure 1 contains an example DFD for a part of our banking system. When logging in, the customer (represented by the *Browser* entity) transmits his or her username and password to the *Load balancer* process. They are then forwarded to the *Login* process, which verifies whether these match with the credentials stored in the *Customer store*. If so, a session is opened in the *Session store* and the session id is sent back to the browser. Subsequently, the customer performs a transaction

by providing the session id and the transaction details (i.e. the source and target accounts, the desired amount, and an optional comment), which are processed by the *Transaction processor* and eventually stored in the *Transaction store*.

DFDs and security. DFDs are more suitable to model the functional data flows of a software system, rather than its security-specific aspects such as user authentication. This is because DFDs lack a unique way to augment them with security-relevant information. For example, in the diagram in Figure 1, the fact that the *Login* process does not compare the actual password of the user to the stored password, but works with hashed versions of them, can only be derived from the smart data flow labels given by the designer. Furthermore, while the designer may know that the *Transaction processor* must authorise an incoming transaction, e.g. check whether the customer is allowed to initiate the transaction from the given source account, the DFD itself does not indicate this.

Despite their successful application in security analysis, this small example already highlights some shortcomings of DFDs. With our model we attempt to alleviate these by providing a formal, security-oriented underpinning that enables the precise definition and verification of security solutions.

III. A PRECISE MODEL FOR SECURITY

From a bird's-eye view, our model consists of three main concepts: *data* operated on by *processes* that can be connected to each other to form *networks*; further detailed in this section.

We have chosen to specify our entire model using the Coq Proof Assistant [10], yielding machine-checked proofs as well as a virtually unconstrained expressivity. For the purpose of comprehensibility and brevity, we opt to limit the amount of Coq code in this paper. The interested reader can consult the complete implementation via the accompanying website [9].

A. Data

Data is often a primary asset in software security, thus representing it is a necessity for our model. We define data as an inductive type (lines 1-10), where each constructor corresponds to a type of data on which processes can operate.

```
1  Inductive Data : Type :=
2  | plain: nat → nat → Data
3  | key: CryptoKey → Data
4  | id: Identity → Data
5  | cred: Credential → Data
6  | sid: SessionId → Data
7  | enc: Data → CryptoKey → Data
8  | hashed: Data → Data
9  | sig: Data → CryptoKey → Data
10 | collection: nat → list Data → Data
```

The simplest, non-security, data type is plain (line 2), where two natural numbers serve as identifiers, allowing to distinguish different data categories. For example, the first identifier `3` categorises `plain 3 9` as a transaction, while the second identifier `9` differentiates it from other transactions.

In the context of security, several types of data are instrumental in reasoning about properties. Therefore, we define these as separate types encapsulating their details, while also providing general data constructors (lines 3-6) allowing to treat them as ordinary data elements. For example, cryptographic keys are defined by the `CryptoKey` data type. This data type encapsulates the different types of keys such as symmetric, private or public keys. For example, `symk 1` represents a symmetric key with `1` as identifier. Similarly we provide data types for an `Identity`, `Credential` and `SessionId`.

Data can be operated on by applying (security) transformations. Our model currently supports three security-related transformations: encrypting data using a cryptographic key, calculating a hash value and digitally signing data using a cryptographic key. Our model also provides constructors for the data resulting from each of these transformations (lines 7-9). For example, the result of encrypting `plain 3 9` using cryptographic key `symk 1` is represented as `enc (plain 3 9) (symk 1)`. Furthermore, several data elements can be collected to construct more complex data structures (line 10).

Note that due to the inductive nature of our data type, each transformation result can again be transformed. For example, hash values can be encrypted just as any other data element.

B. Process

As part of our model, we define a number of processes, each encapsulating a well-defined, possibly non-deterministic behaviour. Since these processes deliver the bulk of the semantics of our model, they serve as the designer's building blocks to construct security designs. Internally, a process consists of a (possibly non-deterministic and infinite) state machine, and a number of input and output queues. Individual process instances are identified by a natural number.

The state machine of a process fully governs its input and output behaviour. This means it determines when to read data from a queue, the operations to perform on this data, and when to write the result to a queue. Processes have access to an unlimited number of input and output queues, from which data can respectively be read and written in a first-in-first-out fashion. Within each process, a queue is identified by a natural number, allowing processes to assign different responsibilities to their queues. The specific processes offered by our model can be roughly divided into three types: security processes, external processes, and auxiliary processes.

Security processes. These processes encapsulate a single security-related operation (Table I). For example, an encrypter encrypts its input data using a provided cryptographic key. To achieve this, an encrypter assigns two queues to key management, one queue via which keys can be provided and one via which the current key can be revoked. The remaining queues of the process serve as input queues for the data to be encrypted and output queues for the encrypted result.

External processes. An application typically interacts with external entities. In our model, these are modelled using external processes (Table II). A user and attacker process respectively represent a non-malicious and malicious user. Both contain a set of data (i.e. knowledge) that they can output. Furthermore, they can learn each piece of data received via interactions with other processes. Additionally, an attacker can derive new information from the knowledge it already

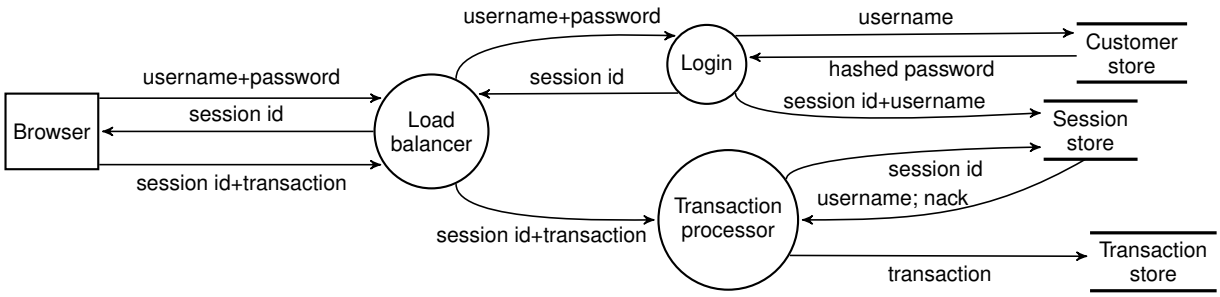

Fig. 1. This traditional DFD of a part of our example banking system is security-specific only because of well-chosen labels.

TABLE I
THE SECURITY PROCESSES DEFINED IN OUR MODEL.

Process	Description
Hasher	Calculates a hash value of its input data.
Encrypter	Encrypts input data with a cryptographic key.
Decrypter	Decrypts input data with a cryptographic key.
Authenticator	Verifies whether an identity and credential match with a looked-up version.
Enforcer	Enforces input to be cleared before passing on.
Authoriser	Encapsulates an authorisation policy by non-deterministically allowing or denying requests.
Generator	Generates a digital signature given a data element and a cryptographic key.
Verifier	Verifies whether data and signature match.

has, and spontaneously obtain new data (i.e. the attacker guesses data from the point of view of the system). By default, both derivation and guessing are specified in an omnipotent manner, so an attacker can derive any data from any other data and guess any data. Consequently, these abilities should be constrained by external assumptions when proving properties.

A source and sink represent entities which respectively provide data to or consume data from other processes. They can represent users, external systems or internal parts of the system for which any further details can be abstracted away.

TABLE II
EXTERNAL PROCESSES REPRESENT ENTITIES THAT INTERACT WITH THE SYSTEM UNDER DESIGN.

Process	Description
User	Non-malicious user interacting with the system.
Attacker	Malicious user interacting with the system.
Source	Produces data satisfying a pre-defined function.
Sink	Consumes its input.

Auxiliary processes. To connect the security and external processes into a functional design model, we need a number of auxiliary processes (Table III). For example, the business process can, non-deterministically, perform any non-security operation. This allows to abstract away any non-security functionality without the need of formalising this functionality. Another example, a store process represents persistent storage, such as a file system or database, and records data as key-value pairs. Each input and output queue of a store process has, based on its identifier, a single predefined responsibility: read a value corresponding to a key, writing a key-value pair, deleting a key-value pair, or listing all used keys.

TABLE III
AUXILIARY PROCESSES TO CONSTRUCT FUNCTIONAL SECURITY DESIGNS.

Process	Description
Business	Encapsulates non-security related functionality.
Store	Stores data as key-value pairs.
Comparator	Compares two data elements using a function.
Collector	Collects the first data element of its input queues.
Disperser	Disperses a collection into its contained elements.
Dropper	Non-deterministically forwards or discards its input.
Discarder	Discards input if directed to by another process.
Joiner	Outputs data from a non-deterministically selected input queue.
Copier	Copies its input to each of its output queues.
Fork	Outputs input to a non-deterministically selected output queue.
Latch	Remembers its last input and continues to output it.

C. Network

Processes can communicate with each other by organising them in a network, thus allowing a designer to model complex systems. In a network, channels connect an output queue of one process to an input queue of another process, allowing the former to communicate its outputs to the latter. Each channel starts at exactly one output queue and ends at exactly one input queue. While this may seem a severe restriction, it can be circumvented using disperser, copier or fork processes to emulate multiple outgoing channels, and collector or joiner processes to emulate multiple incoming channels.

Note that channels are very simple elements, and propagate all data instantaneously without failure. More interesting channels can still be modelled by interposing one or more (auxiliary) processes. For example, a lossy communication channel between two processes can be modelled by placing a dropper between them, instead of directly connecting them.

Since a network is a collection of processes and channels, the state of a network as a whole is the collection of all process states, i.e. their internal state combined with the contents of their queues. There exists a transition between two networks if the second network can be obtained from the first network by sequentially performing the following actions: (1) for each process, perform a local state transition or do nothing, i.e. skip;

and (2) propagate the contents of some or all output queues along the connected channels

Using this transition relation, an infinite sequence of successive networks, called a path in our model, can be constructed. Thus a `path s n` captures one possible execution of the modelled system as a sequence `s` having `n` as initial network.

IV. Modelling a Security Design

In this section, we show how designers can model a broad range of behaviour by organising processes into networks. Figure 2 shows an example network (using an ad-hoc graphical notation, the full implementation can be found online [9]), corresponding to the partial banking system in Figure 1.

Functionality: A minimal amount of functionality must be modelled, before we are able to reason about the security properties of our banking application. The customer's web browser is modelled as a user process, which initially knows the username and password of the customer and a transaction he or she wants to execute. For simplicity we assume that the customer is already registered at the bank with his or her username and password. More specifically, the customer store is instantiated containing a corresponding key-value entry. The load balancer is the entry point to the banking system, and is modelled as a fork process that directs incoming requests to the right process. Note that any other functionality of the load balancer, such as distributing the load over multiple instances, is abstracted away. Finally, the business process makes abstraction of actually executing the transactions.

Simplified HTTPS: In our banking application, the traffic between the customer's web browser and the system is encrypted. In Figure 2, all data sent by the user or load balancer is first encrypted by their respective encrypter processes, and decrypted again upon arrival. The key exchange mechanism of the underlying protocol is modelled by source processes producing symmetric keys and copier processes distributing these keys, and is assumed to be secure. Note that this model does not ensure data integrity, and thus corresponds to a simplified version of the well-known HTTPS protocol.

Authentication: Another security solution is authenticating users using username/password, and (after successful authentication) execute further interactions within a session to avoid resubmitting this data with every request. The actual authentication is modelled using a set of collaborating processes, most notable the authenticator process. Upon arrival of a customer's username and password, the authenticator process requests the credential, if any, stored for this username from the customer store and verifies whether this credential matches the received one. If the credentials match, the authenticator enables two enforcer processes to forward their input (the username and a session id) to the session store, thereby opening a session, and forward the session id to the user process.

Sessions: Session ids are generated by a source process. The user attaches the received session id to its subsequent transaction requests. For each arriving request, the system must now verify whether the corresponding session is still open and, if so, associate the correct identity (i.e. username) with the request for later authorisation. The session id is sent as a read request to the session store, which replies with an acknowledgement and the corresponding username if the session exists. This username is combined with the transaction before being forwarded to the business process. If the session was already closed (e.g. the user logged out), the enforcer will be instructed to discard its input.

Attacker: The security of a system is usually assessed with respect to a type of attacker. Consequently, a security design must explicitly contain an attacker process and the corresponding attacker model. The attacker model is partially reflected by the presence or absence of channels between the attacker and other processes. Since customers connect to the banking system via the Internet, we are (in this example) primarily concerned with external attackers that can tamper with customer requests. This is shown by placing copier, discarder and joiner processes in between the browser and load balancer, and connecting these to the attacker (cf. Figure 2). An attacker can intercept and read (via the copier), modify (via the joiner) or delete (via the discarder) the data that is sent from or to the browser. Furthermore, our attacker model presumes read access to the customer store, e.g. due to a SQL injection vulnerability. In our model, this is modelled by connecting the attacker to a read queue of the customer store. Note that the absence of other channels between the attacker and the internals of the banking system imply that he or she has no further access to the internal communication.

V. Reasoning about security

In this section, we show how the formal semantics of our model can be used to prove security properties about a design.

A. Expressing security properties

A primal condition for proving whether or not certain security properties hold in a design is that these properties themselves are precisely defined. Some common properties are defined as part of our model, and can be readily instantiated and reused. Currently, we express the available properties using linear-time temporal logic (LTL) [11, chapter 3], albeit other formalisms can be added. In LTL, formulas are expressed over infinite sequences of states, called a path. In our model, a network is state, thus a network path as defined earlier can be considered a path over which LTL formulas can be expressed.

A first common property is data confidentiality, defined in line with CNSSI-4009 [12] as *'confidential data is not disclosed to any unauthorised entity'*, where the attacker process acts as unauthorised entity. More precisely, data `d` is confidential in a network `n` if for each `path s n`, an attacker never knows `d` (lines 11–13).

```
11 Definition confidential d n :=
12   ∀ a s, is_attacker a n → path s n →
13   s ⊨ always (¬is_attacker_and_knows a d)
```

Another property is data origin authentication, also called message authentication, defined by CNSSI-4009 [12] as *'the process of verifying that the source of the data is as claimed and that the data has not been modified'*. In terms of our model, this is expressed as follows: each data element `d` that

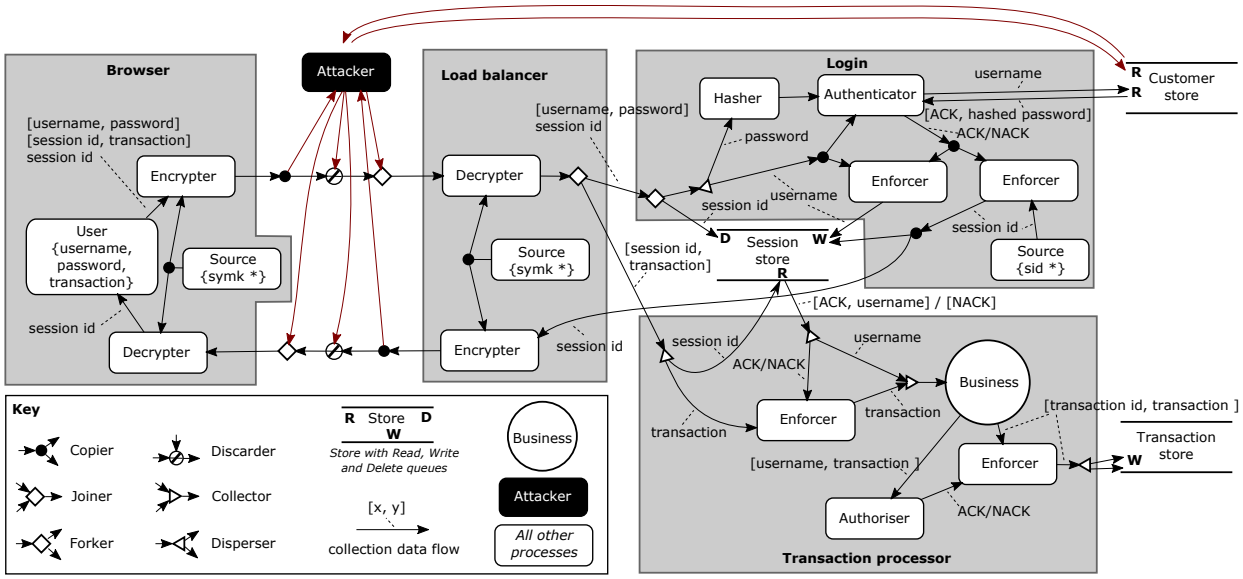

Fig. 2. In contrast to the traditional DFD notation, our model makes security aspects explicit

satisfies a selector function `f`, received via input queue `qr` by process `rcv`, must have been previously sent by the process `snd` via output queue `qs` (lines 14–17). The selector function enables expressing the property for entire categories of data at once, e.g. all transactions in our banking system.

```
14 Definition data_origin_authentication f rcv snd qr
   ↪  qs n :=
15  ∀ s d, path s n → f d →
16  s ⊨ always (contained_in_input d qr rcv →
17       previously (contained_in_output d qs snd)).
```

For our banking application, we require the data origin authentication property with respect to all transactions. That is, each transaction (represented as `plain 3 x` for any `x`) that arrives at the business process should have originated at the user process (cf. Figure 2). Note that this property does not counter so-called replay attacks where an attacker captures a transaction in transit (via the copier in between the browser and load balancer) and re-sends it at a later time. Countering such attacks requires entity authentication, which includes uniqueness and timeliness of arriving data.

B. Proving Security Properties

A proof of a security property can rely on a multitude of already proven theorems that accompany our model. These theorems concern the behaviour of processes as well as networks as a whole, and can be considered an API for writing proofs. For instance, we supply an attacker knowledge theorem proving that an attacker has only four ways of knowing data `d`: (1) `d` is part of its initial knowledge; or the attacker previously (2) guessed `d`; or (3) learned `d`; or (4) derived `d` from other knowledge. Applying this theorem boils down to employing a divide and conquer strategy, where the single hypothesis that the attacker knows `d` is divided into four smaller ones.

As another example, the stepping theorem says that if `d` is found in an input queue of a process, it (1) must have been there initially; or (2) it was previously present in the connected output queue.

Proving data origin authentication for transactions in our banking application boils down to proving that each transaction arriving at the business process was sent by the user process. The following paragraphs provide an informal overview of such a proof; the full proof can be consulted online [9].

We use the above stepping theorem to, starting at the business process, 'step backwards' through our network, and show for each encountered process that a transaction could not have originated there. Most of these steps are fairly mechanical and can be proven using pre-defined theorems.

Less mechanical, hence more interesting, reasoning is required when we arrive at the attacker process. Intuitively we have to contradict that the attacker knows a valid transaction request, i.e. each `enc (collection n [sid x; plain 3 y]) (symk z)` accepted by the system is confidential with respect to the attacker. Using the above attacker knowledge theorem this can be divided into the four ways an attacker can obtain such knowledge.

The most challenging of these four goals is proving that the attacker cannot derive a valid transaction request from all obtainable knowledge. Given the usage of strong encryption, e.g. AES, a reasonable assumption is that encrypted data can only be derived from the corresponding plain text and key. This reduces the goal into two smaller ones, namely either `collection n [sid x; plain 3 y]` or `symk z` must be confidential. The latter would be unrealistic, as it prevents the attacker from knowing any symmetric key.

Proving the confidentiality of `collection n [sid x; plain 3 y]` follows a similar divide and conquer strategy. As before the attacker knowledge theorem gives rise to four smaller goals, of which the derivation case is the most interesting. By introducing an assumption that a collection can only

be derived from all its individual elements, the goal is reduced to proving the confidentiality of `sid x` or `plain 3 y`.

Because we expect an attacker can produce valid transaction data, we focus on the confidentiality of `sid x`. This can be proven based on the network structure and some extra assumptions. For example, an attacker should not be able to guess valid session ids or derive one from other obtained data.

VI. Discussion and Future Work

Currently our model already integrates several security mechanisms and properties into a single coherent model. Yet not all security concepts, e.g. auditing of user actions, are covered. We are confident such concepts can be fairly easily added to our model, yielding a more complete security model.

We intend that in time our model serves as a foundation for a broader security by design approach. Such an approach should readily fit into current software design processes. Since modelling a security design using the Coq Proof Assistant deviates strongly from current notations as DFD or UML, a more natural notation is necessary. The example graphical notation used in this paper is a first step in that direction, but remains at the same level of abstraction as our model itself. A better notation would more closely align with the designer's mindset, by abstracting away the specifics of our model such as the operation-specific queues, and supporting pre-defined, reusable compositions of processes.

Furthermore, designers often re-use the same or similar security solutions in different designs. For example, the use of an encrypted channel (e.g. TLS) between a client and a server is common practice. Within our model, we can formalise such solutions as generic strategies, akin to existing security patterns but modelled in a more precise manner. Such strategies then have to be proven correct only once (under certain assumptions), after which they can be made available in a catalogue and reused as many times as desired. Instead of ensuring that each instance of a security solution is correct, the designer's task is now reduced to verifying whether or not the assumptions made by a strategy are satisfied by the design.

To better support designers, we want to automate verifying assumptions using model checkers. The major challenge here is to correctly encode our semantics as defined in Coq, into the input language for the model checker, and at the same time avoid running into a state explosion problem.

Finally, the benefits of a proven secure design can be drastically increased when the proven guarantees can be transferred to its actual implementation. Efficiently verifying implemented software against its proven design requires further research.

VII. Related work

Over the past decade or so, several approaches to formalise and analyse security designs have been proposed, as attested by several existing surveys in this research area [6], [13], [14].

A prominent approach is UMLsec [15], which introduces UML Machines and UML Machine Systems, similar to processes and networks in our model. They enable formal tools such as automated theorem provers and model checkers to be used to verify security properties of annotated UML models.

Xu and Nygard [16] formalise a design and threats as aspect-oriented petri nets. A design is considered vulnerable when the threat petri net can execute. If so, a mitigating petri net aspect can be woven into the design.

Gürgens et al. [17] describe the behaviour of a system as sets of sequences of actions combined with the knowledge and view agents have about the system. Based on these sequences, security requirements such as authenticity can be specified.

The above approaches incorporate security into the functional description of the system. Whereas our model abstracts away non-security elements of the design where possible.

Heyman et al. [18] formally model several security patterns using Alloy [19], allowing designers to integrate these into their design and evaluating security properties using the assumptions accompanying these patterns. Due to its reliance on Alloy the assumptions are only valid for a finite set of designs, whereas guarantees arising from proofs in our model are valid for all designs. Furthermore, the approach is limited to the formalised patterns and thus less flexible than our model.

Several more specialised approaches focus exclusively on formalising and analysing access control aspects, for example AMF [20], Georg et al. [21], and SecureUML [22]. In contrast, we intend to provide a broad and comprehensive model.

Furthermore, there exists a large amount of techniques to verify cryptographic protocols [23]. While our model is suitable for a rudimentary analysis of such protocols, we do not aim to supplant these specialised techniques but consider them complementary. For example, our model is not intended to discover attacks such as the POODLE attack against SSL.

Aside from security-specific approaches, multiple approaches provide a more formal foundation for software design. Two well-known ones are Communicating Sequential Processes (CSP) [24], focusing on concurrency, and the spi calculus [25], originally intended to analyse security protocols. Our primary reason for not starting from these is flexibility in the definition of our model. By defining our own model, we can fine-tune its semantics without possible limitations of an underlying formalism. Furthermore, the use of the Coq Proof assistant gives us a flexible manner to prove security properties, while automatically verifying the correctness of each proof. It remains to be investigated to what extent our model can be integrated with these calculi.

Several proposals extend DFDs with a formal semantics. For example, France [26] associates a formal semantics, based on algebraic specification, with control-extended DFDs (C-DFD). This extension of DFD introduces, among others, queued flows allowing constructs similar to our collector and disperser processes. Properties such as safety can be verified against a, semi-automatically generated, formal specification.

Alternatively, DFDs have also been enriched with petri net like semantics [27], [28]. There are some similarities with our model. For example, the authors of [28] define bubbles, corresponding roughly to processes in a DFD, that fire in two steps, namely first reading from input flows, and then writing

computed values to output flows. In our model, a process behaves similarly, but it can have multiple intermediate steps. Also, in contrast to the generic bubbles, the processes in our model have pre-defined semantics.

Furthermore, Fraser et al. [29] and Larsen et al. [30] follow an alternative of transforming DFDs to more formal languages, e.g. VDM, allowing to construct formal proofs.

In general, two important distinctions between our model and the above DFD formalisations should be mentioned. First, none of these formalisations are specific to security, in contrast to our security-focused model. Second, with the exception of VDM, they do not support the automatic verification of proofs that our model provides via its implementation in Coq.

VIII. Conclusion

Many stakeholders advocate the principle of security by design, yet designing secure software still is an arduous task. Designers typically have to resort to their own knowledge and experience, possibly augmented with guiding methodologies such as STRIDE and informal catalogues of security solutions.

A very significant challenge for designers is to *correctly instantiate* a security solution; selecting a suitable solution is not the critical challenge. Providing designers with techniques that support them in the correct instantiation of the chosen security solution can obviously improve this situation. Therefore, we proposed a model, inspired by Data Flow Diagrams (DFD), but tailored to security and backed by well-defined semantics.

This paper showed how this model supports the precise expression of and reasoning about the security-relevant aspects of a software design. We provide a formal implementation of our model by using the Coq Proof Assistant. This enables thorough verification of modelled designs by facilitating formal proofs that the applied solutions are indeed correct. Additionally, the use of a proof assistant automatically ensures the correctness of the proofs themselves. Thus our model can complement individual expert knowledge and experience with strong, proven guarantees. Furthermore, we have illustrated our model using a realistically sized banking application and shown how our model enables proving correctness of the applied security solutions, with respect to a security property.

We believe that our model, along with the enabled verification, can serve as a solid basis for the security by design paradigm. It provides a formal foundation enabling designers to reason about security problems and solutions, and on which more extensive and user-friendly modelling and analysis tools can be built.

Acknowledgment

This research is partially funded by the Research Fund KU Leuven and the Secure Design project of the imec HI2 Distributed Trust program.

References

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, Third Edition*. Addison-Wesley, 2013.

[2] C. Steel and R. Nagappan, *Core Security Patterns: Best Practices and Strategies for J2EE", Web Services, and Identity Management*. Pearson Education India, 2006.

[3] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons, 2013.

[4] E. Fernandez-Buglioni, *Security patterns in practice: designing secure architectures using software patterns*. John Wiley & Sons, 2013.

[5] K. Yskout, R. Scandariato, and W. Joosen, "Do security patterns really help designers?" in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE Int. Conf. on*, vol. 1, May 2015.

[6] A. van den Berghe, R. Scandariato, K. Yskout, and W. Joosen, "Design notations for secure software: a systematic literature review," *Software & Systems Modeling*, 2015.

[7] T. DeMarco, *Structured Analysis and System Specification*. Yourdon Press Computing Series, 1978.

[8] A. Shostack, *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[9] Companion website. https://people.cs.kuleuven.be/alexander.vandenberghe/formalise17.html.

[10] The Coq development team, *The Coq proof assistant reference manual*, LogiCal Project, 2016, version 8.6. [Online]. Available: http://coq.inria.fr

[11] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[12] Committee on National Security Systems, *National Information Assurance (IA) Glossary*, May 2003.

[13] P.-H. Nguyen, J. Klein, Y. Le Traon, and M. E. Kramer, "A Systematic Review of Model-Driven Security," in *Software Engineering Conf. (APSEC, 2013 20th Asia-Pacific)*, vol. 1, Dec 2013.

[14] A. V. Uzunov, E. B. Fernandez, and K. Falkner, "Engineering Security into Distributed Systems: A Survey of Methodologies," *j-jucs*, vol. 18, no. 20, dec 2012.

[15] J. Jürjens, *Secure Systems Development with UML*. Springer, 2004.

[16] D. Xu and K. E. Nygard, "Threat-driven modeling and verification of secure software using aspect-oriented petri nets," *IEEE Trans. on Software Engineering*, vol. 32, no. 4, April 2006.

[17] S. Gürgens, P. Ochsenschläger, and C. Rudolph, "On a formal framework for security properties," *Computer Standards & Interfaces*, vol. 27, no. 5, 2005.

[18] T. Heyman, R. Scandariato, and W. Joosen, "Reusable formal models for secure software architectures," in *Working IEEE/IFIP Conf. on Software Architecture (WICSA) and the 6th European Conf. on Software Architecture (ECSA)*, August 2012.

[19] D. Jackson, *Software Abstractions: Resources and Additional Materials*. MIT Press, 2006.

[20] H. Hu and G. J. Ahn, "Constructing authorization systems using assurance management framework," *IEEE Trans. on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 40, no. 4, 2010.

[21] G. Georg, I. Ray, K. Anastasakis, B. Bordbar, M. Toahchoodee, and S. H. Houmb, "An aspect-oriented methodology for designing secure applications," *Information and Software Technology*, vol. 51, no. 5, 2009.

[22] D. Basin, M. Clavel, J. Doser, and M. Egea, "Automated analysis of security-design models," *Information and Software Technology*, vol. 51, no. 5, 2009.

[23] B. Blanchet, "Security protocol verification: Symbolic and computational models," in *Proc. of the First Int. Conf. on Principles of Security and Trust*, ser. POST'12, 2012.

[24] A. W. Roscoe, *Understanding concurrent systems*. Springer Science & Business Media, 2010.

[25] M. Abadi and A. D. Gordon, "A calculus for cryptographic protocols: The spi calculus," in *Proc. of the 4th ACM Conf. on Computer and Communications Security*, 1997.

[26] R. B. France, "Semantically extended dataflow diagrams: a formal specification tool," *Software Engineering, IEEE Trans. on*, vol. 18, no. 4, 1992.

[27] P. D. Bruza and T. P. Van der Weide, "The semantics of data flow diagrams," in *Proc. of the Int. Conf. on Management of Data*, 1989.

[28] G. T. Leavens, T. Wahls, and A. L. Baker, "Formal semantics for SA style data flow diagram specification languages," in *Proc. of the 1999 ACM Symposium on Applied Computing*, ser. SAC '99, 1999.

[29] M. D. Fraser, K. Kumar, and V. K. Vaishnavi, "Informal and formal requirements specification languages: bridging the gap," *IEEE Trans. on Software Engineering*, vol. 17, no. 5, May 1991.

[30] P. G. Larsen, N. Plat, and H. Toetenel, "A formal semantics of data flow diagrams," *Formal aspects of Computing*, vol. 6, no. 6, 1994.